Thread Synchronization

ComS 587X
Fall, 2002

Thread Control

- Interrupting
  - thread.interrupt() method
    - Causes InterruptedException in thread
- Stopping
  - thread.stop()
    - Terminates a thread
    - Can leave data in inconsistent state

Thread Control (2)

- Suspend & Resume
  - thread.suspend()
  - thread.resume()
    - Can result in deadlock
- Yield
  - Thread.yield()
- Sleep
  - Thread.sleep(int ms)

Synchronization

- Allows programmer to maintain data consistency in multi-threaded program
- Mechanisms in Java language
  - Mutual exclusion of synchronized operations on an object
    - Method-level synchronization
    - Block-level synchronization
- Compare to mutexes, condition variables, and semaphores in POSIX C libraries

Thread Conflicts

- Most operations are not atomic
  - E.g., `a = a + 1;` is multiple instructions:
    - read `a` into a register from memory
    - read constant `1` into a register
    - add two registers, leaving result in a register
    - store register to memory
  - Pre-emptive thread switch can occur at any time
  - If two or more threads interleave instructions from `a=a+1`, result is inconsistent

Data Consistency

- Protect operations on common data
  - Generally need to synchronize all read and write operations
- But, synchronization limits concurrency
  - In the worst case, can completely serialize access to objects and negate benefits of multi-threading
  - Minimize the amount of time spend in synchronized code

Method Synchronization

- Instance methods marked “synchronized”
- When a thread enters a synchronized method, the object becomes locked by the thread & access to the object's synchronized methods from other threads is blocked
- Thread can then perform operations on object as though they were “atomic”
- Exiting the method unlocks the object
- No effect on static or non-synchronized methods

Method Synchronization (2)

- **Synchronized** keyword:

```
public class SomeClass {
  protected SomeData sd;
  public synchronized void updateData(...) {
  ...
  }
  public synchronized void getData(...) {
  ...
  }
  public SomeClass(...) {
  }
}
```

Sample Method Synchronization: Counter

```
public class Counter {
  private int countValue;
  public Counter() { countValue = 0; }
  public Counter(int start) { countValue = start; }
  public synchronized void increaseCount() {
    int count = countValue;
    try { Thread.sleep(5); } catch
   (InterruptedException ie) {}
    count = count + 1;
    countValue = count;
  }
  public synchronized int getCount() {
    return countValue; } }
```

Sample: CountingThread

```
public class CountingThread implements Runnable {
  Counter myCounter;  int countAmount;
  public CountingThread(Counter c, int a) {
    myCounter = c;  countAmount = a;
  }
  public void run() {
    for (int i = 1; i <= countAmount; i++)
      myCounter.increaseCount();
  }
  public static void main(String args[]) throws Exception {
    Counter c = new Counter();
    Runnable runner = new CountingThread(c, 10);
    Thread t1=new Thread(runner); Thread t2=new Thread(runner);
    Thread t3=new Thread(runner); Thread t4=new Thread(runner);
    t1.start(); t2.start(); t3.start(); t4.start();
    t1.join(); t2.join(); t3.join(); t4.join();
    System.out.println("Counter value is " + c.getCount());
  }
}
```

Block Synchronization

- Preface a block of statements with the synchronized keyword and an object to protect
- Allows wrapper classes to protect methods on objects that are not thread-safe, E.g.

```
public class Unsafe {
  public void set() {...}
}
public class Safe { Unsafe u;
  public void set() {
    synchronized (Unsafe u) { u.set(); }
}
```

- Apparently can't be used on primitive variables

Summary

- Thread control
- Thread synchronization
  - Why needed
  - Method-level
  - Block-level